

Estruturas de Dados e Algoritmos

Capítulo 3

Conceitos fundamentais de algoritmia e algoritmos de ordenação e pesquisa



3. Conceitos fundamentais de algoritmia e algoritmos de ordenação e pesquisa

- Noção de algoritmo
- Conceção de algoritmos
- Eficiência de algoritmos – notação de O grande
- Algoritmos de ordenação e análise de complexidade
- Algoritmos de pesquisa e análise de complexidade
- Tabelas indexadas
- Ordenação de estruturas de dados complexas usando tabelas indexadas

1ª Aula



Algoritmo

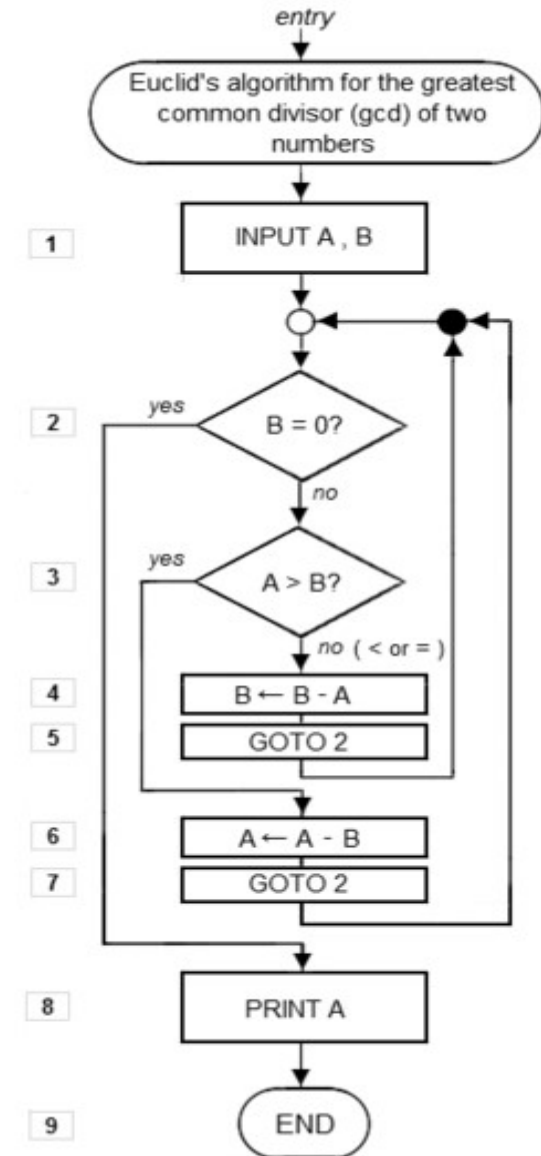
- Do latim *algorithmi*.
- Palavra com origem no apelido Al-Khwarizmi de um matemático persa (IX d.C.):
 - As suas obras foram traduzidas para latim no séc. XII;
 - O título de uma delas foi "***Algorithmi de Numero Indorum***" (algoritmos usando o sistema de numeração decimal).
- Primeiro algoritmo:
 - Mais antigo do que a palavra algoritmo!...
 - Algoritmo de Euclides (300 a.C.): máximo divisor comum de dois números inteiros.





Algoritmo

- Primeira definição formal por Alan Turing (1936).
- Definições informais:
 - "...conjunto de regras que define de forma precisa uma sequência de operações ...eficaz...", H. Stone (1972).
 - Conjunto de instruções que, de uma forma não ambígua, especificam a resolução de um determinado problema.



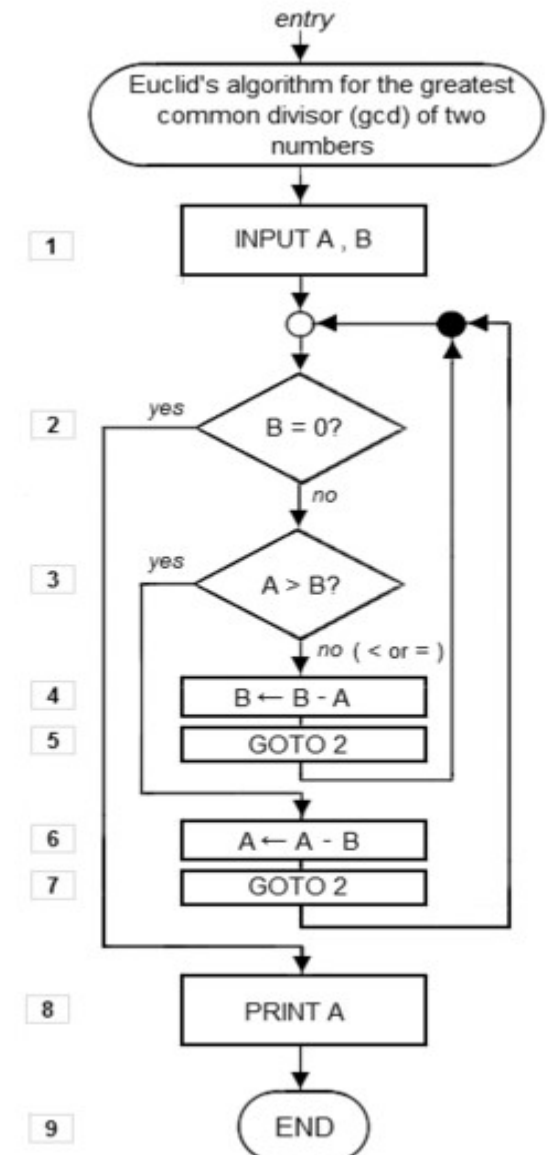


Algoritmos – Alguns Conceitos

- A descrição de um algoritmo deve ser abreviada e clara.
- Utilização de **pseudo-código**.
 - Algures entre a linguagem natural e a linguagem de programação.


Algoritmo de Euclides m.d.c.(A, B)

1. **Entradas:** A, B
2. **Enquanto** $B \neq 0$ **Faz**
3. **Se** $A \leq B$
4. $B \leftarrow B - A$
5. **Senão**
6. $A \leftarrow A - B$
7. **Fim Enquanto**
8. **Retorna:** A





Algoritmos – Alguns Conceitos

- Um algoritmo pode ter várias formas de implementação ao ser escrito numa linguagem de programação.
- Um problema pode ser resolvido por diferentes algoritmos: necessário avaliar quais os mais eficientes
- **Análise de Complexidade de Algoritmos** 
- Provar a eficácia do algoritmo: Está correto? Resolve mesmo o problema em todas as situações?
- Avaliar a eficiência do algoritmo:
Quantidade de recursos computacionais necessários, e.g., tempo de execução, espaço de memória, etc.
- **Estimar recursos exigidos pelo algoritmo em função da dimensão dos dados de entrada**

↑ Recursos ⇔ ↓ Eficiência



Conceção de Algoritmos

- **Especificar o problema**
- **Identificar soluções**
- **Primeira abordagem:**
 - Inputs
 - Outputs
 - Dados (vars a usar)
 - Operações principais
- **Abordagem top-down (programação descendente):**
 - Começa-se por definir as funções gerais (que vão chamando outras de nível mais baixo)
 - Definem-se a seguir as funções de nível a seguir
 - Repete-se o processo até às funções atómicas (nível + baixo)
 - Desvantagem: testes ao software só se podem realizar no final



Conceção de Algoritmos

- **Abordagem bottom-up (programação ascendente):**
 - Começa-se por definir as funções atómicas (nível + baixo)
 - Definem-se a seguir as funções de nível a seguir (superior)
 - Repete-se o processo até às funções principais
 - Vantagem: testes ao software pode ser realizado incrementalmente
- **Abordagem mista (combina top-down com bottom-up):**
 - Definem-se as funções gerais (que vão chamando outras de nível mais baixo)
 - Definem-se a seguir as funções chave, que resolvem aspetos particulares ou fundamentais do problema
 - Definem-se a seguir as restantes funções (de níveis superiores e de níveis inferiores)



Análise de Complexidade de Algoritmos

- **Complexidade espacial** de um algoritmo
 - Espaço de memória que necessita para executar até ao fim.
 - Espaço de memória exigido em função do tamanho N dos dados.
- **Complexidade temporal** de um algoritmo
 - Tempo que demora a executar (tempo de execução).
 - Tempo de execução em função do tamanho N dos dados.
- \uparrow Complexidade \Leftrightarrow \downarrow Eficiência



Análise de Complexidade de Algoritmos

- Na grande maioria dos casos, é praticamente impossível prever com exatidão o tempo de execução de um algoritmo ou de um programa.
- **Solução:**
 - Procura-se **identificar as operações mais frequentes e/ou mais demoradas** e determina-se o **n° de vezes que são executadas**.
 - Para se obter uma estimativa do tempo de execução do algoritmo, procura-se **determinar uma expressão que relacione esse tempo com o tamanho N dos dados** de entrada.
 - Constantes multiplicativas e parcelas menos significativas são desprezadas, ficando **apenas os termos dominantes**.



Análise de Complexidade de Algoritmos – *Notação de O Grande*

- O resultado é depois expresso utilizando a chamada **notação de O grande** (*Big O Notation*).

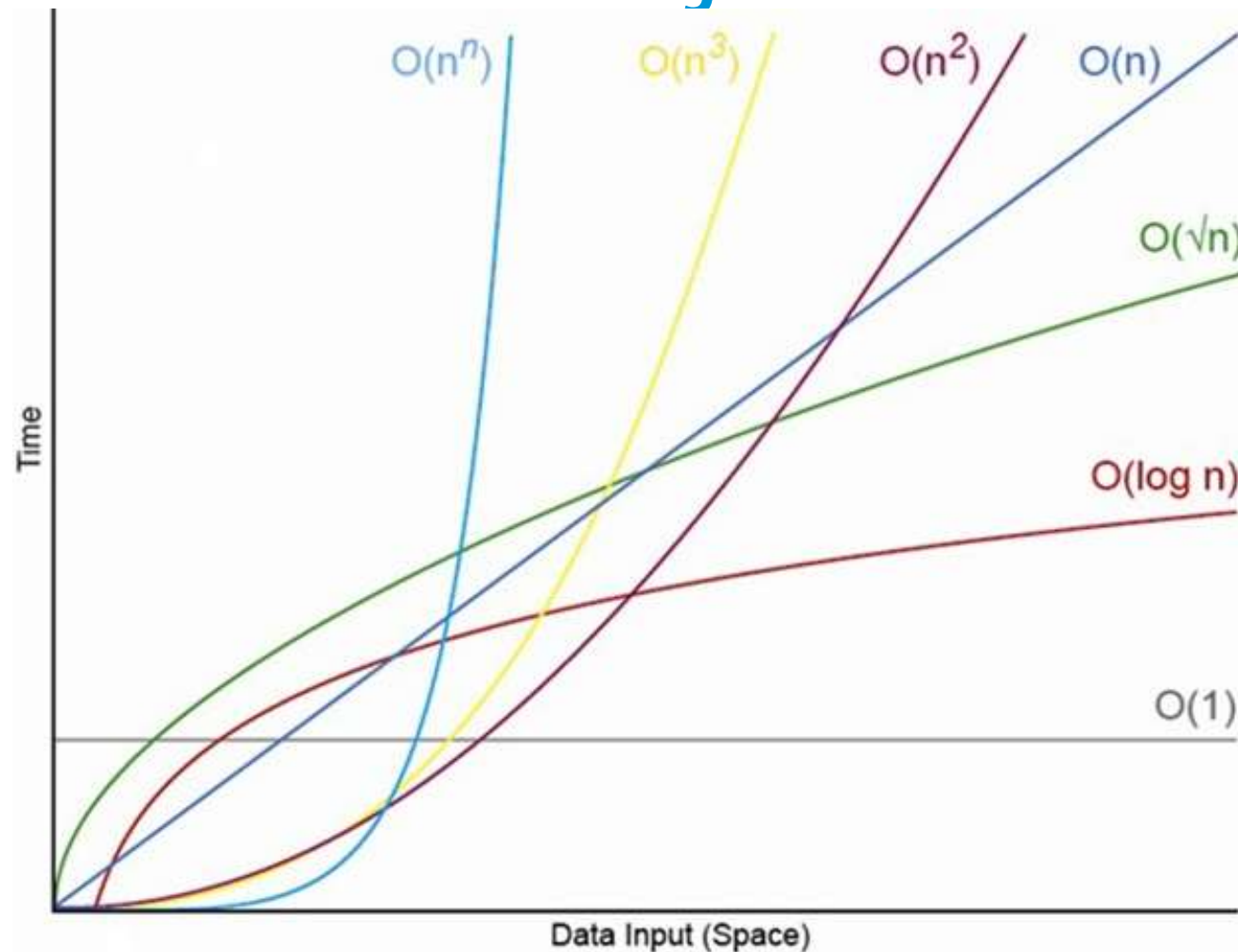
- Exemplos:

- $2^N + 5 \rightarrow O(2^N)$ (ordem exponencial)
- $a_k N^k + a_{k-1} N^{k-1} + \dots + a_0 \rightarrow O(N^k)$ (ordem polinomial)
- $2N^2 + 3 \rightarrow O(N^2)$ (ordem quadrática)
- $N \log N \rightarrow O(N \log N)$ (ordem linear-logarítmica)
- $N + 5 \rightarrow O(N)$ (ordem linear)
- $\log_2 N \rightarrow O(\log N)$ (ordem logarítmica)
- $4 \rightarrow O(1)$ (ordem constante)





Análise de Complexidade de Algoritmos – *Notação de O Grande*





Algoritmos de Ordenação

- A existência de conjuntos de **dados ordenados** é bastante comum na nossa vida quotidiana e, consequentemente, também o é em programas informáticos.
- Exemplos:
 - Listas telefónicas;
 - Pautas de alunos;
 - Listas de produtos de uma empresa;
 - Tabela classificativa;
 - etc.
- É fundamental utilizar-se **algoritmos eficientes** para ordenar grandes quantidades de dados.



Algoritmos de Ordenação

■ Exemplo – Lista Telefónica

```
struct registo { // campos de vários tipos
    char Nome[101];
    char Morada[201];
    char Telefone[10];
};
```

Tabela



Registo



Chave



Nome	Morada	Telefone
José Manuel	Rua da Torre, 34	255123456
Manuel Santos	Av. da Fonte, 55	255546783
Carlos Silva	Praça da Igreja, 12	255875645
Óscar Antunes	Urb. do Pinhal, Lt. 2-3B	255123746



Algoritmos de Ordenação

- **Tipos** de ordenação:
 - Ordenação interna é aquela realizada na **memória** principal do computador.
 - Ordenação externa é aquela onde os registos podem estar numa memória auxiliar (e.g. arquivo em **disco**).



Algoritmos de Ordenação

(*Sorting algorithms*)

- **Algoritmos básicos** de ordenação – complexidade $O(N^2)$:
 - Ordenação por **Seleção** (*Selection Sort*)
 - Ordenação por «**Bolha**» (*Bubble Sort*)
 - Ordenação por **Inserção** (*Insertion Sort*)
 - Ordenação Rápida (*Quicksort*) – complex. $O(N \log N)$ no melhor caso
- **Algoritmos avançados** de ordenação – $O(N \log N)$ ou $O(N)$:
 - *Heapsort* - Robert W. Floyd e J.W.J Williams, 1964, $O(N \log N)$
 - *Shell Sort* - Donald Shell, 1959, $O(N \log N)$ até $O(N)$)
 - *Radix Sort* - Harold H. Seward, 1954, $O(nk)$, n é o nº de chaves, k o comprimento médio da chave



Ordenação por Seleção

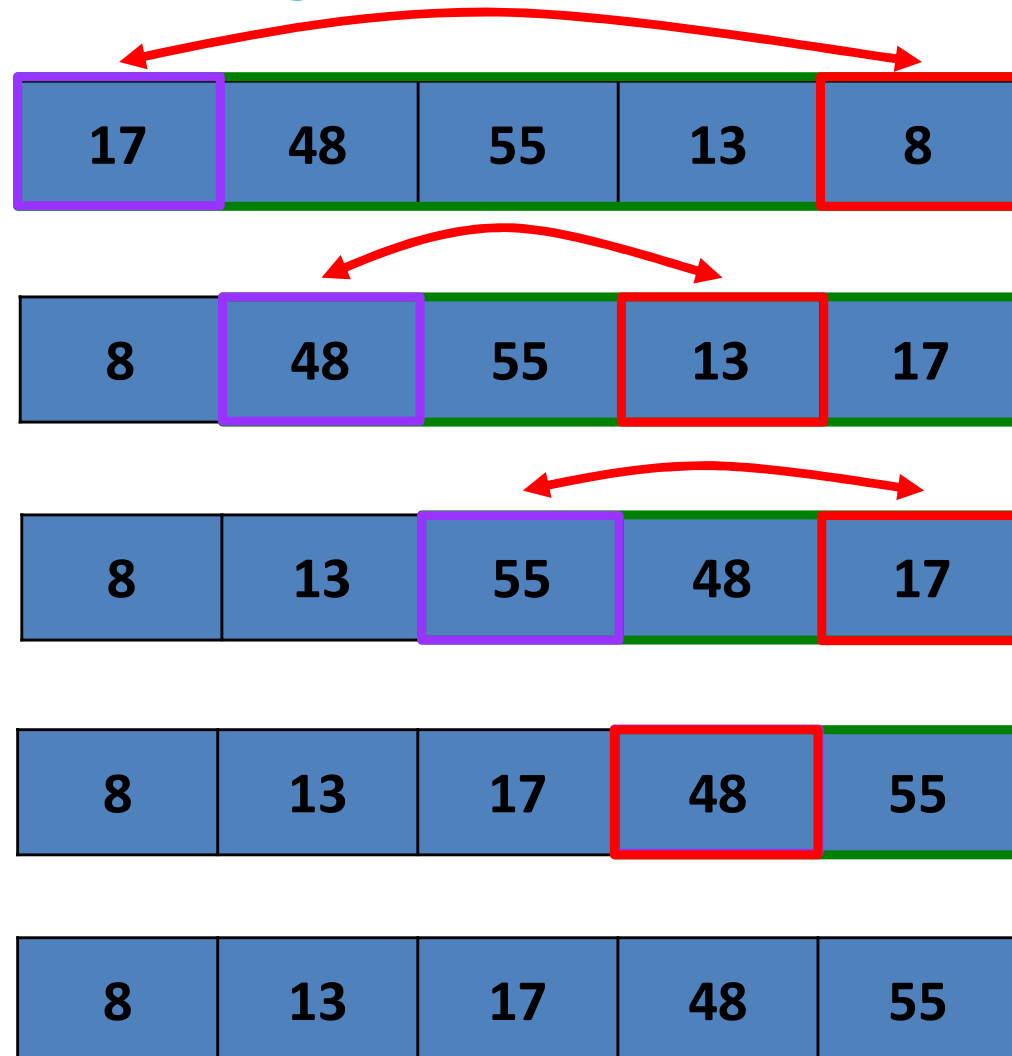
- Provavelmente o algoritmo de ordenação mais intuitivo:
 - Encontrar o mínimo do vetor;
 - Trocar de posição o primeiro elemento com o mínimo do vetor;
 - Continuar para o resto do vetor, procedendo de forma idêntica para o 2º elemento, o 3º elemento, etc., até ao elemento $n-1$, excluindo o(s) elemento(s) já ordenado(s).

Ordenação por Seleção

1. **Entradas:** vetor v de dimensão n
2. **Para Todos** elementos i de 1 até $n-1$ **Faz**
3. $iMenor \leftarrow i$
4. **Para Todos** elementos j de $i+1$ até n **Faz**
5. **Se** $v[j] < v[iMenor]$ **Então** $iMenor \leftarrow j$
6. **Fim Para Todos** j
7. Troca de posição os elems i e $iMenor$
8. **Fim Para Todos** i
9. **Retorna:** vetor v ordenado



Ordenação por Seleção





Ordenação por Seleção



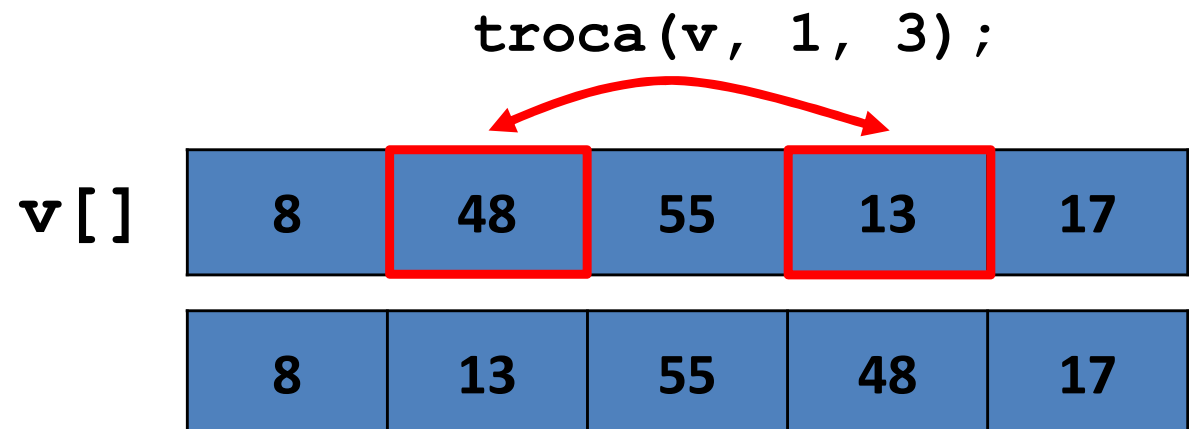
Fonte: <http://www.sorting-algorithms.com>



Ordenação por Seleção

- A função `troca(int v[], int i, int j)` troca de posição o elemento `i` do vetor `v` com o elemento `j`.

```
void troca(int v[], int i, int j) {  
    int aux = v[i];  
    v[i] = v[j];  
    v[j] = aux;  
}
```



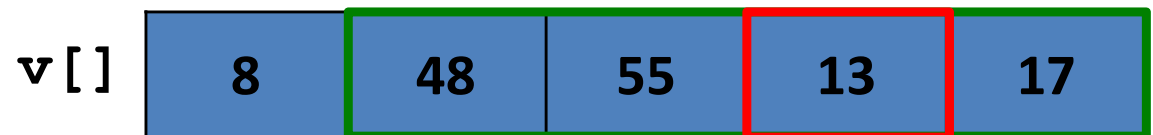


Ordenação por Seleção

- A função `indiceMenor(int v[], int ini, int fim)` determina o índice do menor elemento do vetor `v` situado entre os índices `ini` e `fim`.

```
int indiceMenor(int v[], int ini, int fim) {  
    int iMenor=ini;  
    for(int i=ini+1; i<=fim; i++)  
        if(v[i] < v[iMenor]) iMenor=i;  
    return(iMenor);  
}
```

`a=indiceMenor(v,1,4);`



`a ← 3`



Ordenação por Seleção

- Código C/C++ do algoritmo:

```
// ordena por seleção vetor v com n elementos  
void ordenacaoSelecao(int v[], int n) {  
    for (int i = 0; i < n-1; i++)  
        troca(v, i, indiceMenor(v, i, n-1) );  
}
```

ordenacaoSelecao(v, 5);

v[]

8	48	55	13	17
---	----	----	----	----

8	13	17	48	55
---	----	----	----	----



Ordenação por Seleção

- Análise de complexidade do algoritmo:
 - O número de comparações para um vetor de N elementos é:

$$\begin{array}{rcl}
 \text{Comp} & = & (N-1) + (N-2) + \dots + 2 + 1 \\
 + \text{Comp} & = & 1 + 2 + \dots + (N-2) + (N-1)
 \end{array}$$

$$2 * \text{Comp} = N + N + \dots + N + N$$

$$2 * \text{Comp} = N * (N-1)$$

$$\text{Comp} = \frac{N(N-1)}{2} = O(N^2)$$



Ordenação por «Bolha»

(Bubble Sort)

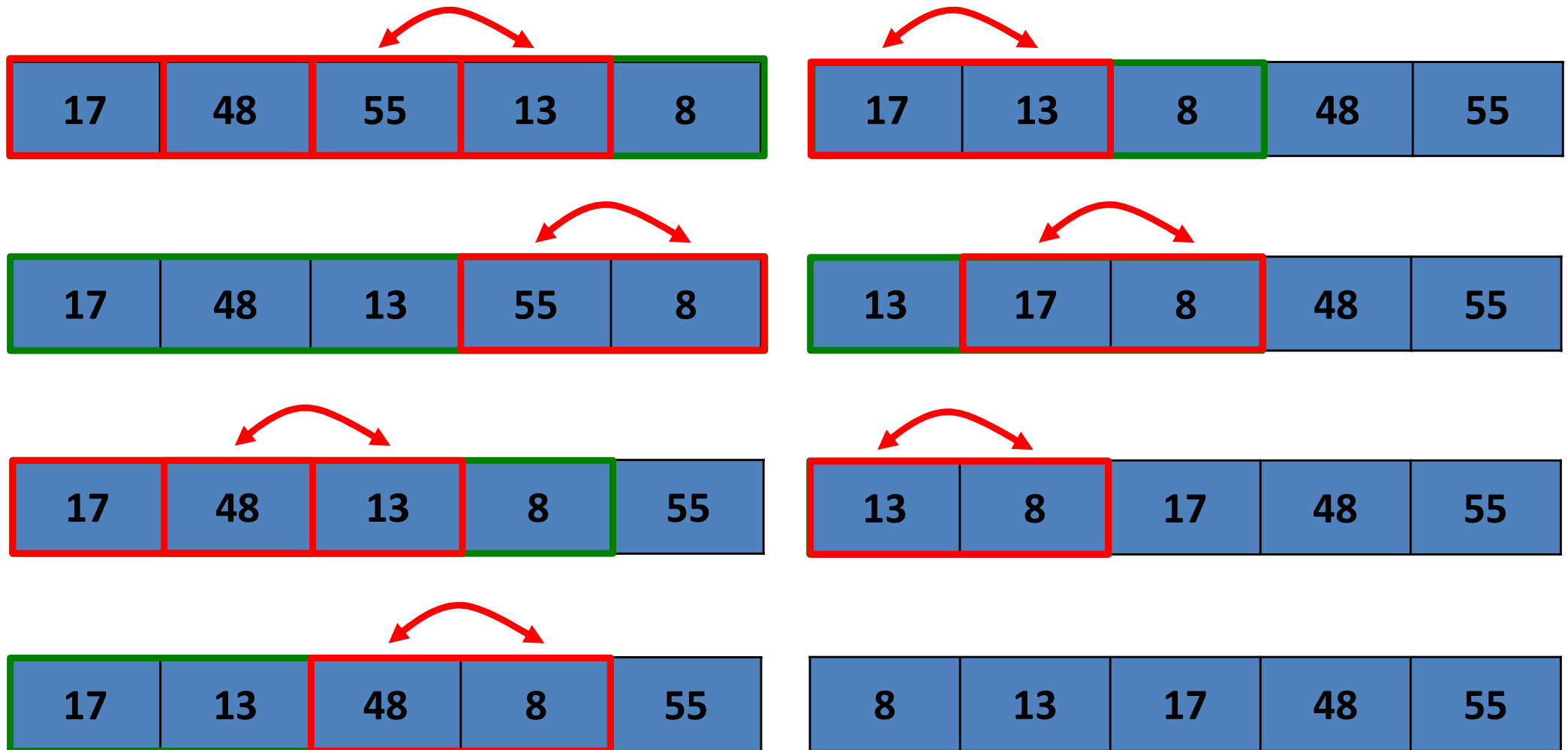
- Compara elementos adjacentes: se desordenados, troca-os.
- Faz isto do 1.º até ao último par.
- Repete, usando menos um par em cada iteração, até não haver mais pares ou trocas.
 - Na 2.ª iteração, processa-se todos os elementos exceto o último. **Porquê?**
 - O algoritmo pode terminar se numa iteração não ocorrer qualquer troca.
- Várias variantes...

Ordenação por «Bolha»

1. **Entradas:** vetor v de dimensão n
2. $\text{flag} \leftarrow 1$
3. **Para** i de 0 até $n - 2$ e **Se** $\text{flag} = 1$ **Faz**
4. $\text{flag} \leftarrow 0$
5. **Para** elementos j de 2 até $n - i$ **Faz**
6. **Se** $v[j - 1] > v[j]$ **Então**
7. Troca de posição elems $j - 1$ e j
8. $\text{flag} \leftarrow 1$
9. **Fim Para** elementos j
10. **Fim Para** i
11. **Retorna:** vetor v ordenado



Ordenação por «Bolha»





Ordenação por «Bolha»



Fonte: <http://www.sorting-algorithms.com>



Ordenação por «Bolha»

■ Código C/C++ do algoritmo:

```
void ordenacaoBolha(int v[], int n) {  
    bool desordenado = true; // inicializa flag  
    for (int i = 0; i < n-1 && desordenado; i++) {  
        desordenado = false; // ordenado se sem trocas  
        for (int j = 1; j < n - i; j++) {  
            if (v[j] < v[j-1] ) { // se fora de ordem  
                troca(v, j-1, j); // troca e...  
                desordenado = true; // ainda desordenado  
            }  
        }  
    }  
}
```



Ordenação por «Bolha»

- Análise de complexidade do algoritmo:
 - O número de comparações para um vetor de N elementos é:

$$Comp = (N-1) + (N-2) + . . . + 2 + 1$$

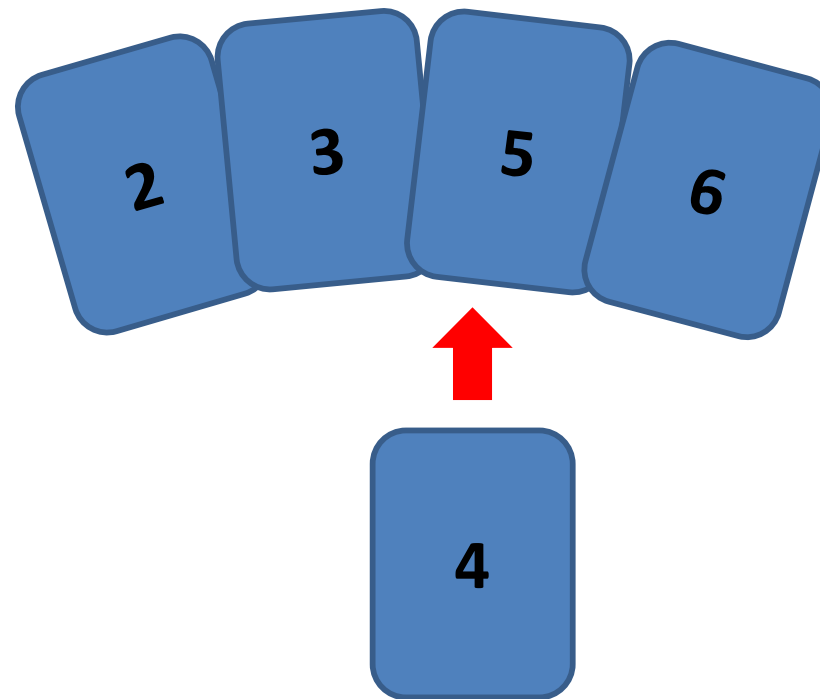
$$Comp = \frac{N(N-1)}{2} = O(N^2)$$

- Igual número de comparações mas menor número de trocas:
 - Algoritmo de seleção: # trocas = $N-1$;
 - Algoritmo por «bolha»: # trocas = entre 0 e o número de comparações.



Ordenação por Inserção

- Funciona de forma idêntica à ordenação das cartas na mão do jogador, durante um jogo de cartas.





Ordenação por Inserção

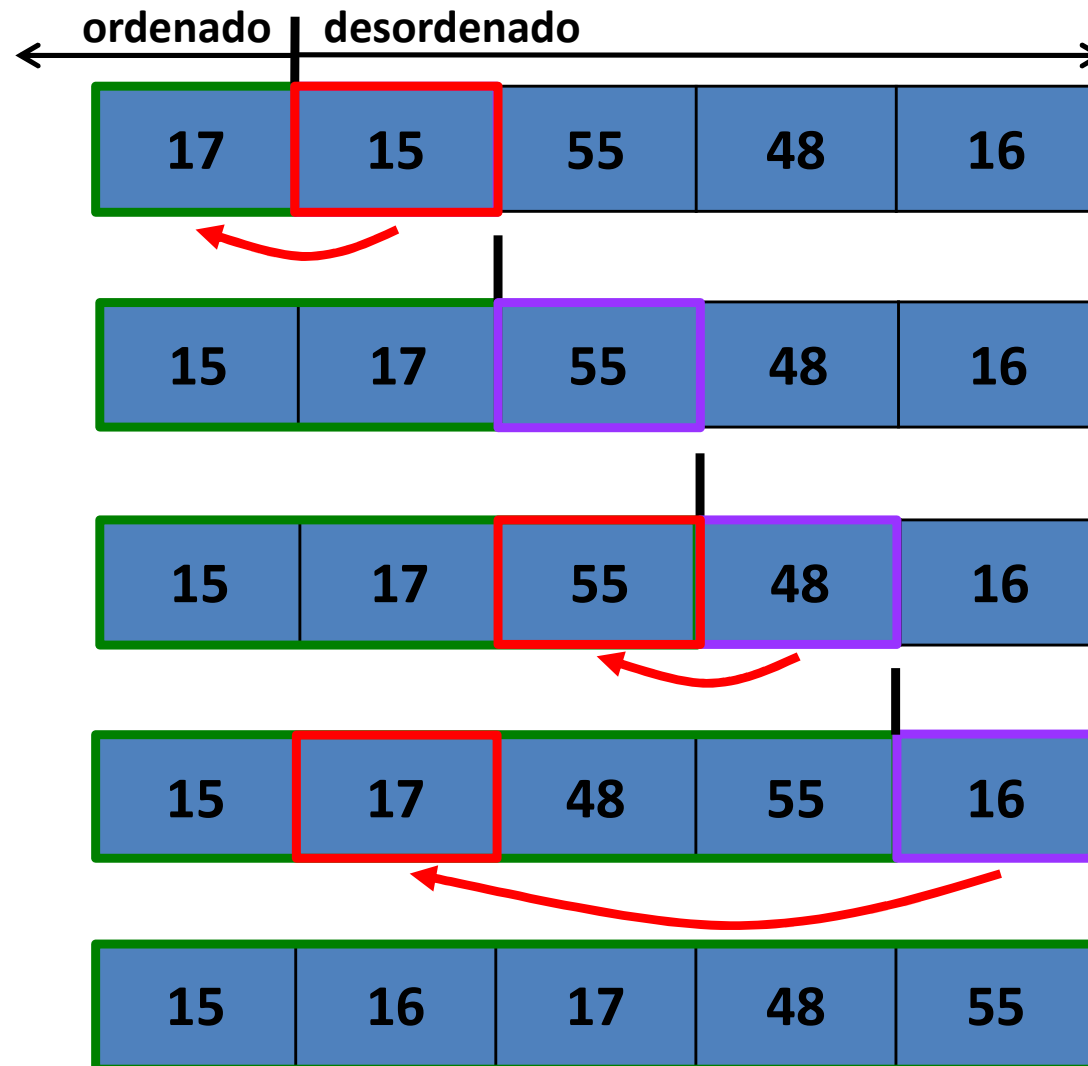
- Considera-se o vetor dividido em 2 sub-vetores, esquerdo e direito, com o da esquerda ordenado e o da direita desordenado.
- Começa-se com apenas um elemento no sub-vetor da esquerda.
- $N-1$ passos:
 - Move-se um elemento de cada vez do sub-vetor da direita para o sub-vetor da esquerda, inserindo-o na posição correta.
 - Termina-se quando o sub-vetor da direita fica vazio.

Ordenação por Inserção

1. **Entradas:** vetor v de dimensão n
2. **Para Todos** elementos i de 2 até n **Faz**
3. $\text{aux} \leftarrow v[i]$
4. **Para** elementos j de i até 2, para baixo e enquanto $\text{aux} < v[j - 1]$ **Faz**
5. $v[j] \leftarrow v[j - 1]$
6. **Fim Para** elementos j
7. $v[j] \leftarrow \text{aux}$
8. **Fim Para Todos** i
9. **Retorna:** vetor v ordenado



Ordenação por Inserção





Ordenação por Inserção



Fonte: <http://www.sorting-algorithms.com>



Ordenação por Inserção

■ Código C/C++ do algoritmo:

```
// ordena por inserção vetor v com n elementos
```

```
void ordenacaoInsercao(int v[], int n) {  
    for (int i = 1; i < n; i++) { // n-1 iterações  
        int aux = v[i]; // elemento a inserir  
        int j; // vai conter índice onde inserir  
            // necessário declarar antes do for  
        for (j = i; (j > 0) && (aux < v[j-1]); j--)  
            v[j] = v[j-1]; // desloca elemento  
        v[j] = aux; // inserção  
    }  
}
```



Ordenação por Inserção

- Análise de complexidade do algoritmo:
 - Dois ciclos encaixados que podem ter até N iterações.
 - O pior caso é para um vetor ordenado pela ordem inversa, em que o número de comparações para um vetor de N elementos é:

$$Comp = 1 + 2 + \dots + (N-2) + (N-1)$$

$$Comp = \frac{N(N-1)}{2} = O(N^2)$$



Ordenação Rápida (*Quicksort*)

- Princípio: *dividir para conquistar*.
- Passos do algoritmo (recursivo):
 1. Caso básico: se o tamanho N do vetor for 0 ou 1, v já está ordenado.
 2. Passo de Partição:
 - Escolher um elemento x do vetor v para funcionar como pivô.
 - Formar 2 sub-vetores a partir de v , com valores $\leq x$ à esquerda do pivô e valores $\geq x$ à direita do pivô.
 3. Passo recursivo: ordenar os sub-vetores esquerdo e direito, usando o mesmo método recursivamente.
- Passo de Partição:
 - Escolher para pivô, x , o elemento do meio do vetor v .
 - Inicializar $i=1$ (índice 1º elem.) e $j=n$ (índice último elemento).
 - Enquanto $i \leq j$ fazer:
 - Enquanto $v[i] < x$, incrementar i .
 - Enquanto $v[j] > x$, decrementar j .
 - Se $i \leq j$ então trocar $v[i]$ e $v[j]$, incrementar i e decrementar j .
 - O sub-vetor esquerdo é $v[1], \dots, v[j]$ (vazio se $j < 1$).
 - O sub-vetor direito é $v[i], \dots, v[n]$ (vazio se $i > n$).



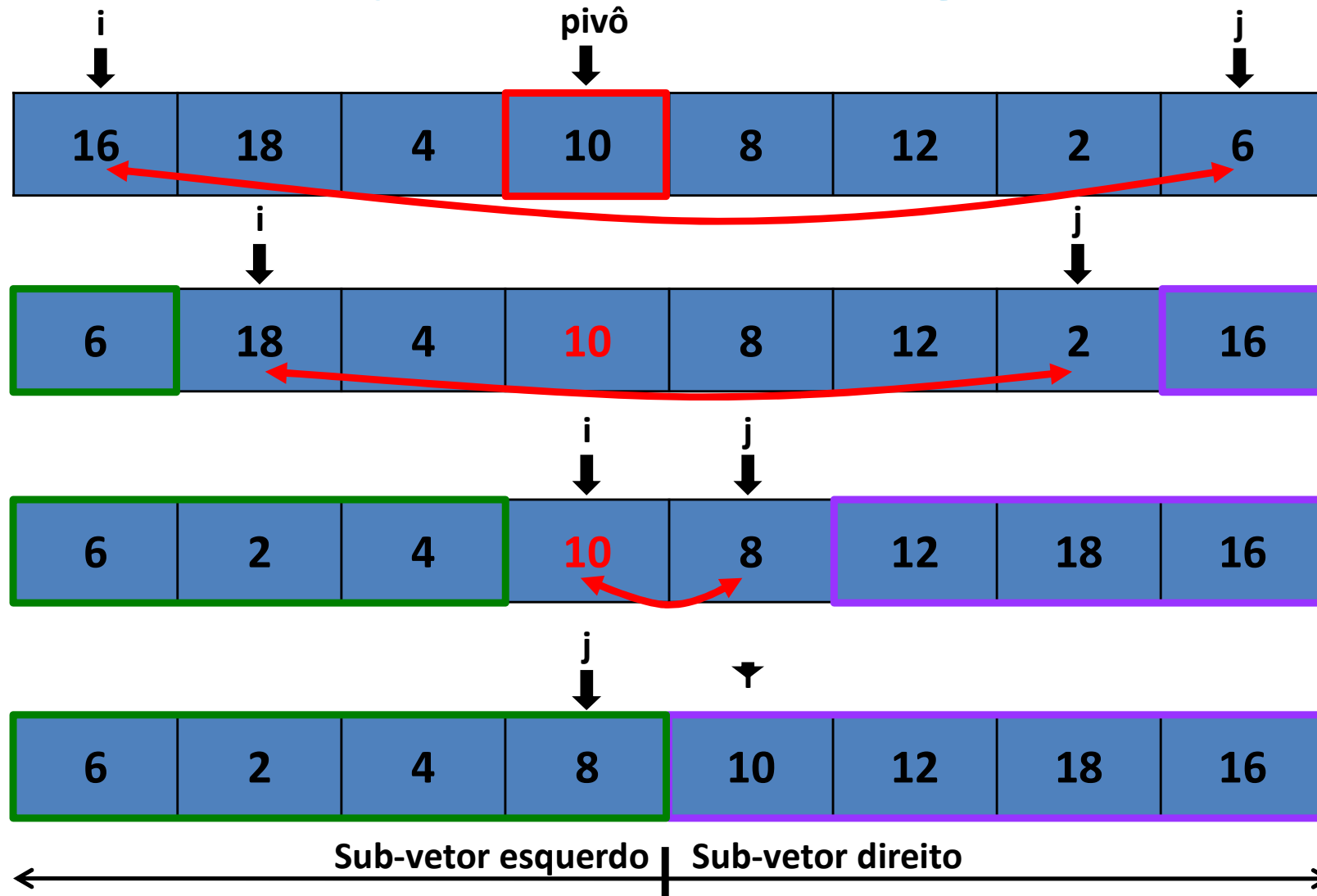
Ordenação Rápida (*Quicksort*)

Ordenação Quicksort

1. **Entradas:** vetor v de dimensão n e índices $i1$ e $i2$ do 1º e último elementos
2. **Se** $i1 \geq i2$ **Então** salta para linha 13
3. $\text{pivot} \leftarrow v[(i1+i2)/2]$; $i \leftarrow i1$; $j \leftarrow i2$
4. **Faz**
5. **Enquanto** $v[i] < \text{pivot}$ **Faz** $i \leftarrow i + 1$
6. **Enquanto** $v[j] > \text{pivot}$ **Faz** $j \leftarrow j - 1$
7. **Se** $i \leq j$ **Então**
8. Troca de posição elementos i e j
9. $i \leftarrow i + 1$ e $j \leftarrow j - 1$
10. **Enquanto** $i \leq j$
11. Ordena sub-vetor $v[i1] \dots v[j]$ (sub-vetor esquerdo)
12. Ordena sub-vetor $v[i] \dots v[i2]$ (sub-vetor direito)
13. **Retorna:** vetor v ordenado



Ordenação Rápida (*Quicksort*)





Ordenação Rápida (*Quicksort*)



Fonte: <http://www.sorting-algorithms.com>



Ordenação Rápida (*Quicksort*)

■ Código C/C++ do algoritmo:

```
void ordenacaoQuicksort(int v[], int inicio, int fim) {  
    if (inicio >= fim) return;    // caso elementar (tamanho vetor <=1)  
    int pivot = v[(inicio + fim) / 2];    // escolhe pivô ("pivot")  
    int i = inicio;  
    int j = fim;  
    do {  
        while (v[i] < pivot) i++;  
        while (v[j] > pivot) j--;  
        if (i > j) break;    // termina as trocas se i "ultrapassou" j  
        troca(v, i++, j--);  
    } while (i <= j);    // quando termina, j < i  
    ordenacaoQuicksort(v, inicio, j);    // ordena recursiv subvetor esq  
    ordenacaoQuicksort(v, i, fim);    // ordena recursiv subvetor dir  
}
```



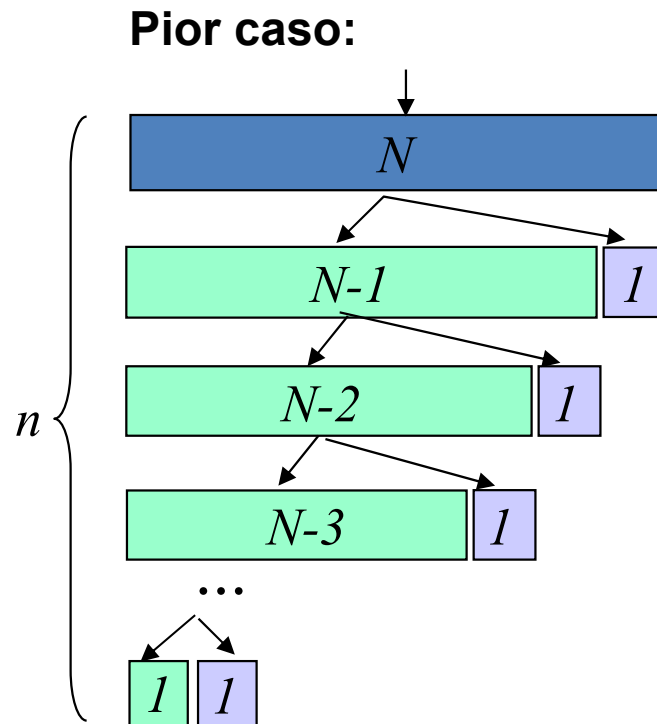
Ordenação Rápida (*Quicksort*)

- Análise de complexidade do algoritmo:
 - O número de comparações efetuadas no processo de **partição** em cada chamada à função é igual ao tamanho do vetor, N , porque todos os elementos são comparados com o pivô.
 - A complexidade do passo de **partição** é portanto $O(N)$.
 - Para determinar o tempo total de execução do algoritmo é necessário somar os tempos de execução do passo de partição em cada uma das chamadas recursivas da função (próximo slide).

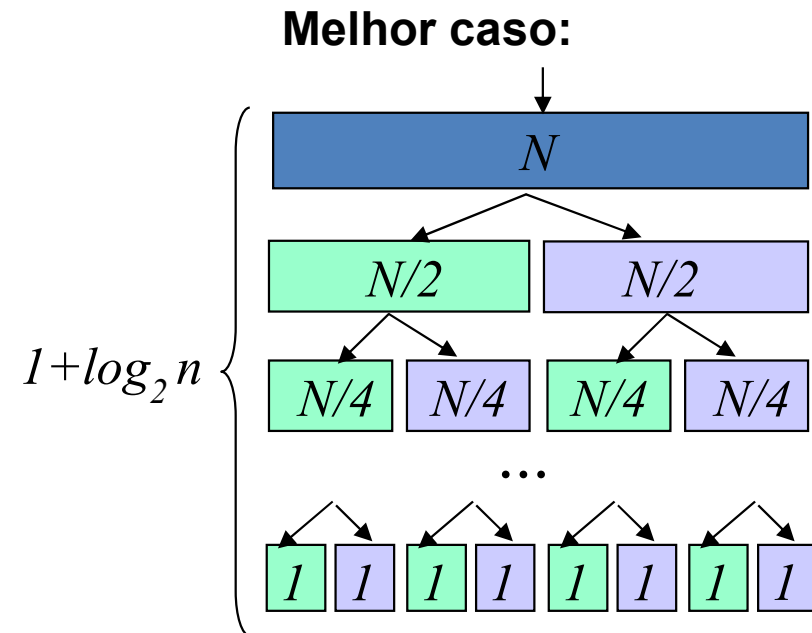


Ordenação Rápida (*Quicksort*)

- Número de chamadas recursivas para um vetor de tamanho N :



$$\begin{aligned}
 T(N) &= cN + T(N-1) = c[N + (N-1) + (N-2) \dots + 3] + T(2) \\
 &= c[N + (N-1) + \dots + 2] = c(N+2)(N-1)/2 \\
 T(N) &= O(N^2)
 \end{aligned}$$



$$\begin{aligned}
 T(N) &= cN + 2T(N/2) = cN + 2[cN/2 + 2T(N/4)] \\
 &= cN \log_2 N + N \\
 T(N) &= O(N \log N)
 \end{aligned}$$



Ordenação Rápida (*Quicksort*)

- Escolha do pivô mais criteriosa:
 - Para otimizar a eficiência do algoritmo pode-se utilizar melhores estratégias na escolha do elemento pivô.
 - Objetivo: possibilitar a partição do vetor em 2 sub-vetores com tamanhos iguais.
 - Exemplos:
 - Mediana de 3 elementos, dos 2 extremos e do ponto médio.
 - Elemento médio dos K primeiros elementos do vetor.
 - Escolher aleatoriamente K elementos do vetor, ordená-los e selecionar o elemento central.
- Desvantagem: tempo de computação extra que é necessário para a seleção do pivô.



3. Conceitos fundamentais de algoritmia e algoritmos de ordenação e pesquisa

- Noção de algoritmo
- Conceção de algoritmos
- Eficiência de algoritmos – notação de O grande
- Algoritmos de ordenação e análise de complexidade
- Algoritmos de pesquisa e análise de complexidade
- Tabelas indexadas
- Ordenação de estruturas de dados complexas usando tabelas indexadas

2ª Aula



Ordenação de Estruturas de Dados

- Ex.: ordenação por seleção de um vetor de estruturas

```
typedef struct {  
    char nome[51];  
    int idade;  
    double salario;  
} Pessoa;  
//-----  
Pessoa v[100];  
//-----  
void ordenacaoSelecao(Pessoa v[], int n) {  
    for (int i = 0; i < n-1; i++)  
        troca( v, i, indiceMenor(v, i, n-1) );  
}
```




Ordenação de Estruturas de Dados

- Ex.: orden. por seleção de vetor de estruturas (cont.)

```
void troca(Pessoa v[ ], int i, int j) {  
    Pessoa aux;  
    aux = v[i];  
    v[i] = v[j];  
    v[j] = aux;  
}  
//-----  
int indiceMenor(Pessoa v[ ],int i1, int i2) {  
    int iMenor = i1;  
    for(int i=i1+1; i<=i2; i++)  
        if(strcmp(v[i].nome, v[iMenor].nome) < 0) iMenor = i;  
    return(iMenor);  
}
```



Ordenação de Estruturas de Dados

- Os algoritmos de ordenação apresentados implicam a reordenação de toda a estrutura de dados que compõem as tabelas.
- Mover estruturas de dados complexas pode ser muito pesado computacionalmente.
- As tabelas apenas são ordenadas por um dos seus campos.
 - E se for necessário disponibilizar a informação usando ordenações distintas? Por ex. por nome e por idade.
- Solução: **Tabelas Indexadas**



Tabelas Indexadas

**Tabela de índices
inicialmente ordenada
por índices**

0	Pedro	36
1	Tiago	22
2	André	44
3	João	38
4	Filipe	29

**Ordenação por
Nome**

2
4
3
0
1

Pedro	36
Tiago	22
André	44
João	38
Filipe	29

**Ordenação por
Idade**

1
4
0
3
2



Tabelas Indexadas

- Ordenação de um vetor de estruturas de diferentes formas:

```
//-----  
typedef struct {  
    char nome[51];  
    int idade;  
    double salario; // podemos ter muitos mais campos!  
} Pessoa;  
  
Pessoa pess[5]={ {"Pedro",36,1000.0}, {"Tiago",22,700.0},  
                 {"André",44, 1100.0}, {"João",38,800.0},  
                 {"Filipe",29,900.0} };    // Dados  
  
int iPorNome[5]; // tabela (vetor) de índices  
int iPorIdade[5]; // tabela (vetor) de índices
```



Tabelas Indexadas

- Ordenação por inserção – por ordem alfabética:

```
// ordena o vetor de indices vi[] por ordem alfabética do
// nome da tabela pess de Pessoas
void ordenaInsercaoNome(Pessoa pess[], int vi[], int n) {
    for (int i=1; i<n; i++) {
        int aux = vi[i]; // indice auxiliar
        int j;
        for (j = i;
             j>0 && strcmp(pess[aux].nome, pess[vi[j-1]].nome)<0;
             j--) { vi[j] = vi[j-1]; }
        vi[j] = aux;
    }
}
```



Tabelas Indexadas

- Ordenação por inserção – por ordem alfabética:

```
//-----  
int ne=5;  // numero de elementos  
void mostraOrdenaPorNome() {  
    // inicializa tab de indices - fundamental!  
    for(int i=0; i<ne; i++) iPorNome[i]=i;  
    // ordena tab de índices por nome  
    ordenaInsercaoNome(pess, iPorNome, ne);  
    // mostra tabela pess (de Pessoas) ordenada por nome  
    for(int i=0; i<ne; i++)  
        cout << "Nome: " << pess[iPorNome[i]].nome <<  
            " - Idade: " << pess[iPorNome[i]].idade << endl;  
}  
//-----
```

Nome: André – Idade: 44

Nome: Filipe – Idade: 29

Nome: João – Idade: 38

Nome: Pedro – Idade: 36

Nome: Tiago – Idade: 22



Tabelas Indexadas

- Ordenação por inserção – por idade:

```
//-----  
// ordena o vetor de indices vi[] por ordem crescente de  
// idades da tabela pess de Pessoas  
void ordenaInsercaoIdade(Pessoa pess[], int vi[], int n) {  
    for (int i=1; i<n; i++) {  
        int aux = vi[i]; // indice auxiliar  
        int j;  
        for (j = i; (j > 0) &&  
            (pess[aux].idade < pess[vi[j-1]].idade); j--)  
            vi[j] = vi[j-1];  
        vi[j] = aux;  
    }  
}  
//-----
```



Tabelas Indexadas

■ Ordenação por inserção – por idade:

```
//-----  
void mostraOrdenaPorIdade() {  
    // inicializa vetor de indices IMPORTANTE!  
    for(int i=0; i<5; i++) iPorIdade[i]=i;  
    // ordena vetor de índices por idade  
    ordenaInsercaoIdade(pess, iPorIdade, 5);  
    // mostra tabela pess (de Pessoas) ordenada por idade  
    for(int i=0; i<5; i++)  
        cout << "Nome: " << pess[iPorIdade[i]].nome  
            << " - Idade: " << pess[iPorIdade[i]].idade  
            << endl;  
}  
//-----
```

Nome: Tiago – Idade: 22
Nome: Filipe – Idade: 29
Nome: Pedro – Idade: 36
Nome: João – Idade: 38
Nome: André – Idade: 44



Tabelas Indexadas

- Ordenação de um vetor de estruturas de diferentes formas:

```
//-----  
int main() {  
    char opcao;  
    cout << "Opção de ordenação: (N ou I)" << endl;  
    do  
        cin>>opcao;  
    while ( (opcao!='N') && (opcao!='I') );  
    if (opcao=='N') mostraOrdenaPorNome( );  
    else mostraOrdenaPorIdade( );  
}  
//-----
```



Tabelas Indexadas em Ficheiros

Em memória, →
chave e
posição (offset)

Nome	Posição
Pedro	0
Tiago	1
André	2
João	3
Filipe	4

← Em ficheiro,
dados
completos

André	2
Filipe	4
João	3
Pedro	0
Tiago	1

Pedro	36
Tiago	22
André	44
João	38
Filipe	29

→ Posição $\times \text{sizeof}(\text{Pessoa})$, para ser em bytes



Tabelas Indexadas em Ficheiros

```
//-----  
typedef struct { // dados em ficheiro  
    char nome[51];  
    int idade;  
    double salario; // podemos ter muitos mais campos!  
    ...  
} Pessoa;  
  
typedef struct { // dados em memória  
    char iNome[51]; // nome correspondente ao índice pos  
    int pos; // posição do registo no ficheiro (em bytes)  
} IdxNome;  
  
IdxNome iPorNomeFic[MAX_REGISTOS]; // tabela em memória  
  
fstream fich;
```



Tabelas Indexadas em Ficheiros

```
// le do fich e guarda em memoria nome e posição
// retorna o n.º de elementos lidos
int criaIndexPorNome( ) {
    Pessoa aux;
    int i = 0;

    fich.seekg(0); // vai para o início do fich
    while( !fich.eof() ) {
        fich.read( (char *) &aux, sizeof(aux) ); //le do fich
        strcpy(iPorNomeFic[i].iNome, aux.nome); //guarda em...
        iPorNomeFic[i].pos = i*sizeof(Pessoa); //memoria
        i++;
    }
    return i;
}
```



Tabelas Indexadas em Ficheiros

```
//-----  
void ordenaInsercaoNomeFic (IdxNome v[ ], int n) {  
    for (int i = 1; i < n; i++) {  
        IdxNome aux = v[i];  
        int j;  
        for (j = i;  
            (j>0) && (strcmp(aux.iNome, v[j-1].iNome)<0);  
            j--) {  
            v[j] = v[j-1];  
        }  
        v[j] = aux;  
    }  
}  
//-----
```



Tabelas Indexadas em Ficheiros

```
//-----  
void ordenaImprimePorNome(int numElementos) {  
    Pessoa p;  
    // ordena (por nome) tabela (indexada) em memória  
    ordenaInsercaoNomeFic(iPorNomeFic, numElementos);  
    // vai ler dados ao ficheiro e imprime nome e idade  
    for(int i=0; i < numElementos; i++) {  
        fich.seekg(iPorNomeFic[i].pos);  
        fich.read((char *) &p, sizeof(p));  
        cout << "Nome: " << p.nome;  
        cout << " - Idade: " << p.idade << endl;  
    }  
}
```



Tabelas Indexadas em Ficheiros

```
//-----  
int main() {  
    int numElementos;  
  
    fich.open("Teste.dta", ios::in | ios::binary);  
  
    numElementos = criaIndexPorNome( );  
  
    ordenaImprimePorNome(numElementos);  
  
    fich.close();  
}
```

Função `qsort ()` do C

- Existe na biblioteca `cstdlib` a função `qsort ()` que implementa o algoritmo *Quicksort*. Pode ordenar qualquer tipo de dados.
 - `void qsort(void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));`
 - `base`: ponteiro para o 1.º elemento do vetor;
 - `nelem`: número de elementos do vetor a ordenar;
 - `width`: número de bytes de cada elemento (usar `sizeof ()`);
 - `fcmp`: função de comparação entre dois elementos – função implementada pelo programador de acordo com o tipo de dados do vetor e com o critério de ordenação a utilizar, retornando **<0** se 1.º elemento menor do que o 2.º, **0** se os dois elementos forem iguais, ou **>0** se 1.º elemento maior do que o 2.º.



Função qsort () – Exemplo 1

```
//-----  
#include <cstdlib>    // biblioteca da função qsort()  
using namespace std;  
const int N=6;  
//-----  
int comparaInt(const void *p1, const void *p2) {  
    return ( *(int *)p1 - *(int *)p2 );  
}  
//-----  
int main() {  
    int tab[N] = {35, 21, 20, 16, 19, 25};  
    qsort((void *) tab, N, sizeof(tab[0]), comparaInt);  
}  
//-----
```



Função qsort () – Exemplo 2

```
#include <stdlib.h> // biblioteca da função qsort()
const int N=6;
struct Registo {
    int idade; // idade
    char nome[21]; // nome até 20 caracteres + '\0'
};
//-----
int comparaNome(const void *preg1, const void *preg2) {
    return strcmp( ( (Registo *) preg1 )->nome,
                  ( (Registo *) preg2 )->nome );
}
//-----
int main() {
    Registo tab[N] = {35, "Rui", 21, "Manuel", 20,
                     "Jorge", 16, "Luis", 19, "Pedro", 25, "Artur"};
    qsort((void *) tab, N, sizeof(Registo), comparaNome);
}
```



Algoritmos de Pesquisa

- Objetivo: verificar se um determinado valor existe num vetor (tipicamente de grande dimensão) e, no caso de existir, determinar a sua posição no vetor.
- No caso de existir mais do que uma ocorrência, há várias possibilidades:
 - Indicar a posição da primeira ocorrência (mais habitual);
 - Indicar a posição da última ocorrência;
 - Não importa qual a ocorrência a devolver.

55 ?



Existe na posição 2.

17	15	55	48	8
----	----	----	----	---



Pesquisa Sequencial

- Percorre o vetor sequencialmente, do primeiro ao último elemento, ou ao contrário, até se encontrar o elemento a pesquisar.
- A pesquisa termina quando:
 - Se encontra o elemento pretendido;
 - Se chega ao fim (ou ao início) do vetor.
- Este algoritmo não exige que o vetor esteja ordenado.
- A pesquisa sequencial apenas é viável em vetores de pequena dimensão.



Pesquisa Sequencial

- Código C/C++ do algoritmo:

```
//-----  
// retorna o índice de valor ou -1 se não existe  
int pesquisaSequencial(int v[], int n, int valor) {  
    for (int i = 0; i < n; i++)  
        if (v[i] == valor) return i; //Foi encontrado  
    return -1; // O valor não foi encontrado  
}  
//-----
```

pesquisaSequencial(v, 5, 55);

v[]	17	15	55	48	8
------------	-----------	-----------	-----------	-----------	----------



Retorna 2.



Pesquisa Sequencial

- Eficiência do algoritmo para um vetor de tamanho N :
 - A operação mais frequente é a comparação.
 - N comparações no pior caso, se não encontra o valor ou é o último a ser pesquisado.
 - $N/2$ comparações em média, se encontra o valor.
 - Complexidade do algoritmo em ambos os casos:

$$T(N) = O(N)$$

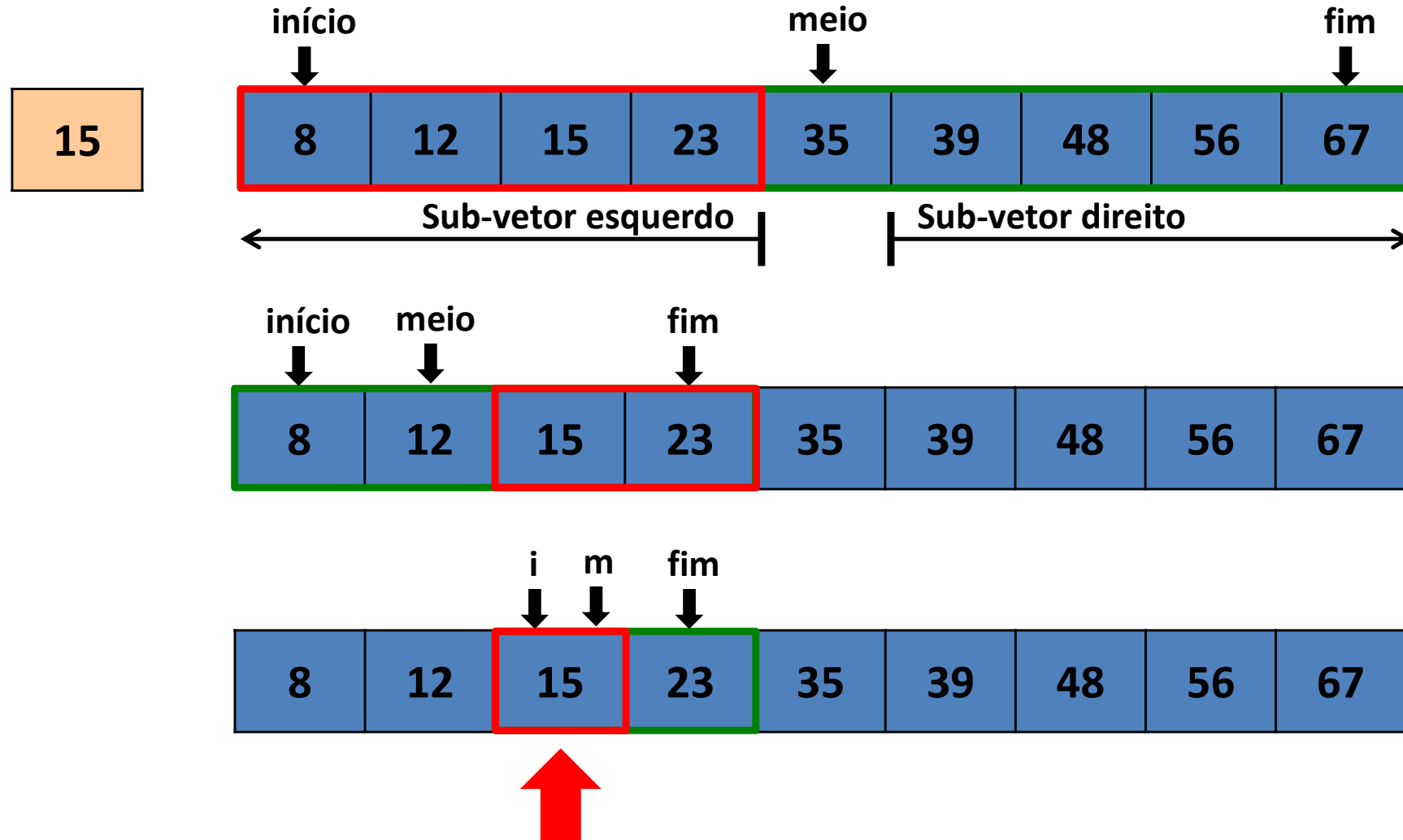


Pesquisa Binária

- Assume que o vetor a pesquisar está ordenado.
- Compara o valor a pesquisar com o elemento situado no meio do vetor:
 - Se forem iguais, o valor foi encontrado e a pesquisa termina.
 - Se o valor a pesquisar for menor do que ele, então repete o processo de pesquisa no sub-vetor de elementos menores do que ele (sub-vetor esquerdo).
 - Se o valor a pesquisar for maior do que ele, então repete o processo de pesquisa no sub-vetor de elementos maiores do que ele (sub-vetor direito).



Pesquisa Binária – Exemplo





Pesquisa Binária

■ Código C/C++ do algoritmo – **Solução Iterativa:**

```
//-----  
int pesquisaBinaria(int v[], int n, int valor) {  
    int inicio = 0, fim = n - 1, meio;  
  
    while (inicio <= fim) {  
        meio = (inicio + fim) / 2;  
        if (valor == v[meio]) return meio; //valor encontrado  
        else if (valor > v[meio]) inicio = meio + 1;  
        else fim = meio - 1; // valor < v[meio]  
    }  
    return -1; // valor não encontrado  
}
```



Pesquisa Binária

■ Código C/C++ do algoritmo – **Solução Recursiva:**

```
//-----  
int pesquisaBinaria(int v[],  
                    int inicio, int fim, int valor) {  
    if (inicio > fim) return -1; //valor não encontrado  
  
    int meio = (inicio + fim) / 2;  
  
    if (valor > v[meio])  
        return (pesquisaBinaria(v, meio+1, fim, valor));  
    if (valor < v[meio])  
        return (pesquisaBinaria(v, inicio, meio-1, valor));  
    return meio; // valor encontrado  
}  
//-----
```



Pesquisa Binária

- Eficiência do algoritmo para um vetor de tamanho N :
 - Em cada iteração, o tamanho do sub-vetor a analisar na iteração seguinte é dividido por um fator de ≈ 2 .
 - Ao fim de k iterações, o tamanho do sub-vetor a analisar é $\approx N/2^k$.
 - No pior caso, o número de iterações é $\approx \log_2 N$:

$$T(N) = O(\log N)$$

- Se o vetor não estiver ordenado à partida, falta contabilizar nesta equação a sua ordenação!



4. Programação Orientada a Objetos

A partir da
próxima aula...